django-oidc-provider Documentation

Release 0.5.x

Juan Ignacio Fiorentino

Oct 17, 2018

Contents

This tiny (but powerful!) package can help you to provide out of the box all the endpoints, data and logic needed to add OpenID Connect capabilities to your Django projects. And as a side effect a fair implementation of OAuth2.0 too. Covers Authorization Code, Implicit and Hybrid flows.

Also implements the following specifications:

- OpenID Connect Discovery 1.0
- OpenID Connect Session Management 1.0
- OAuth 2.0 for Native Apps
- OAuth 2.0 Resource Owner Password Credentials Grant
- Proof Key for Code Exchange by OAuth Public Clients

Before getting started there are some important things that you should know:

- Despite that implementation MUST support TLS, you *can* make request without using SSL. There is no control on that.
- Supports only requesting Claims using Scope values, so you cannot request individual Claims.
- If you enable the Resource Owner Password Credentials Grant, you MUST implement protection against brute force attacks on the token endpoint

---

Contents:

CHAPTER 1

Installation

1.1 Requirements

- Python: `2.7` `3.4` `3.5` `3.6`
- Django: `1.8` `1.9` `1.10` `1.11` `2.0`

1.2 Quick Installation

If you want to get started fast see our `/example` folder in your local installation. Or look at it on github.

Install the package using pip:

```
$ pip install django-oidc-provider
```

Add it to your apps in your project's django settings:

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'oidc_provider',
    # ...
)
```

Include our urls to your project's `urls.py`:

```
urlpatterns = patterns('',
    # ...
```

(continues on next page)

(continued from previous page)

```
    url(r'^openid/', include('oidc_provider.urls', namespace='oidc_provider')),
    # ...
)
```

Run the migrations and generate a server RSA key:

```
$ python manage.py migrate
$ python manage.py creatersakey
```

Add this required variable to your project's django settings:

```
LOGIN_URL = '/accounts/login/'
```

CHAPTER 2

Relying Parties

Relying Parties (RP) creation is up to you. This is because is out of the scope in the core implementation of OIDC. So, there are different ways to create your Clients (RP). By displaying a HTML form or maybe if you have internal trusted Clients you can create them programatically. Out of the box, django-oidc-provider enables you to create them by hand in the django admin.

OAuth defines two client types, based on their ability to maintain the confidentiality of their client credentials:

- `confidential`: Clients capable of maintaining the confidentiality of their credentials (e.g., client implemented on a secure server with restricted access to the client credentials).
- `public`: Clients incapable of maintaining the confidentiality of their credentials (e.g., clients executing on the device used by the resource owner, such as an installed native application or a web browser-based application), and incapable of secure client authentication via any other means.

2.1 Properties

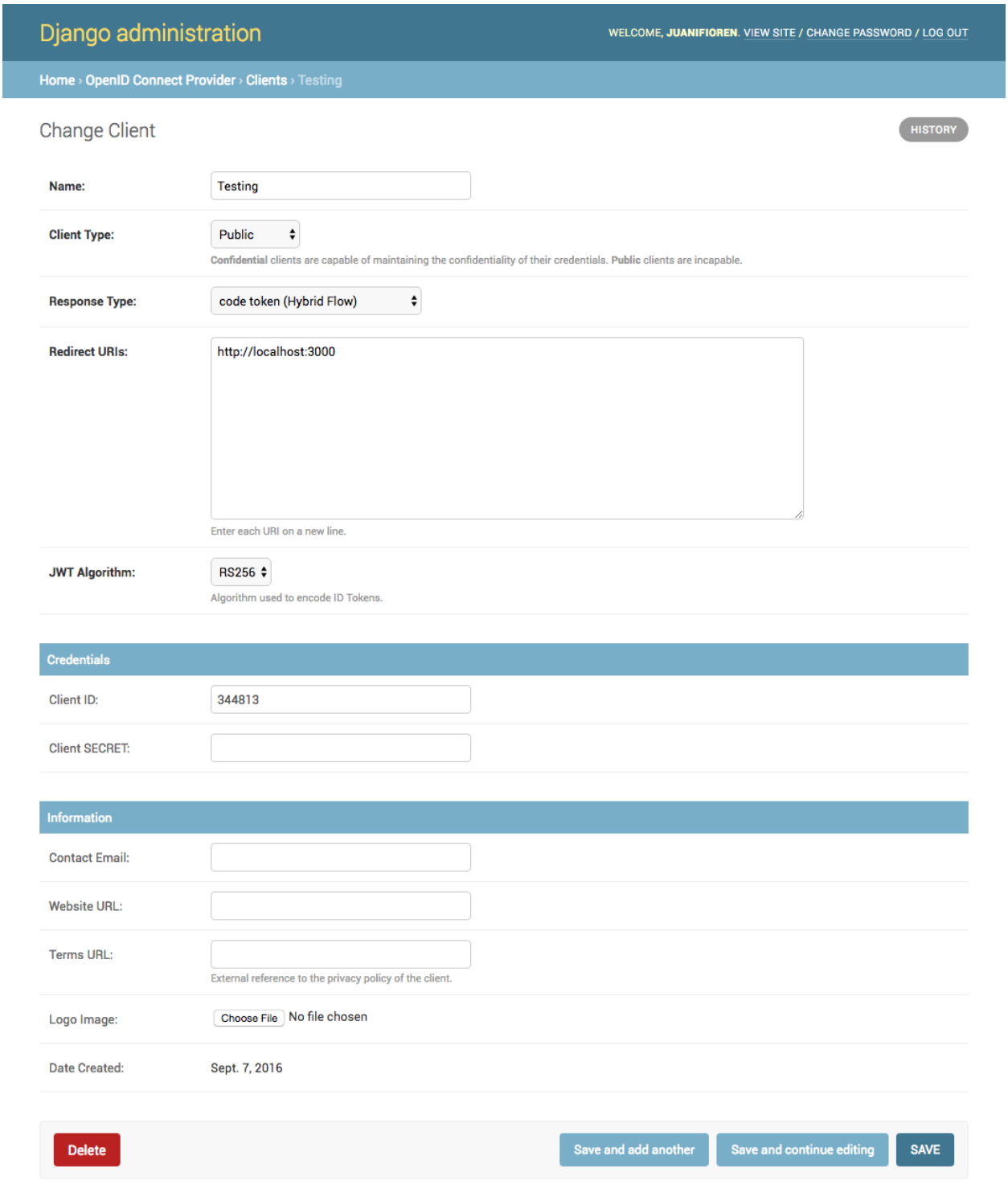
- `name`: Human-readable name for your client.
- `client_type`: Values are `confidential` and `public`.
- `client_id`: Client unique identifier.
- `client_secret`: Client secret for confidential applications.
- `response_types`: The flows and associated `` `response_type` `` values that can be used by the client.
- `jwt_alg`: Clients can choose which algorithm will be used to sign id_tokens. Values are `HS256` and `RS256`.
- `date_created`: Date automatically added when created.
- `redirect_uris`: List of redirect URIs.
- `require_consent`: If checked, the Server will never ask for consent (only applies to confidential clients).
- `reuse_consent`: If enabled, the Server will save the user consent given to a specific client, so that user won't be prompted for the same authorization multiple times.

Optional information:

- `website_url`: Website URL of your client.
- `terms_url`: External reference to the privacy policy of the client.
- `contact_email`: Contact email.
- `logo`: Logo image.

2.2 Using the admin

We suggest you to use Django admin to easily manage your clients:



For re-generating `client_secret`, when you are in the Client editing view, select "Client type" to be `public`. Then after saving, select back to be `confidential` and save again.

2.3 Custom view

If for some reason you need to create your own view to manage them, you can grab the form class that the admin makes use of. Located in `oidc_provider.admin.ClientForm`.

Some built-in logic that comes with it:

- Automatic `client_id` and `client_secret` generation.
- Empty `client_secret` when `client_type` is equal to `public`.

2.4 Programmatically

You can create a Client programmatically with Django shell `python manage.py shell`:

```
>>> from oidc_provider.models import Client, ResponseType
>>> c = Client(name='Some Client', client_id='123', client_secret='456', redirect_
↪uris=['http://example.com/'])
>>> c.save()
>>> c.response_types.add(ResponseType.objects.get(value='code'))
```

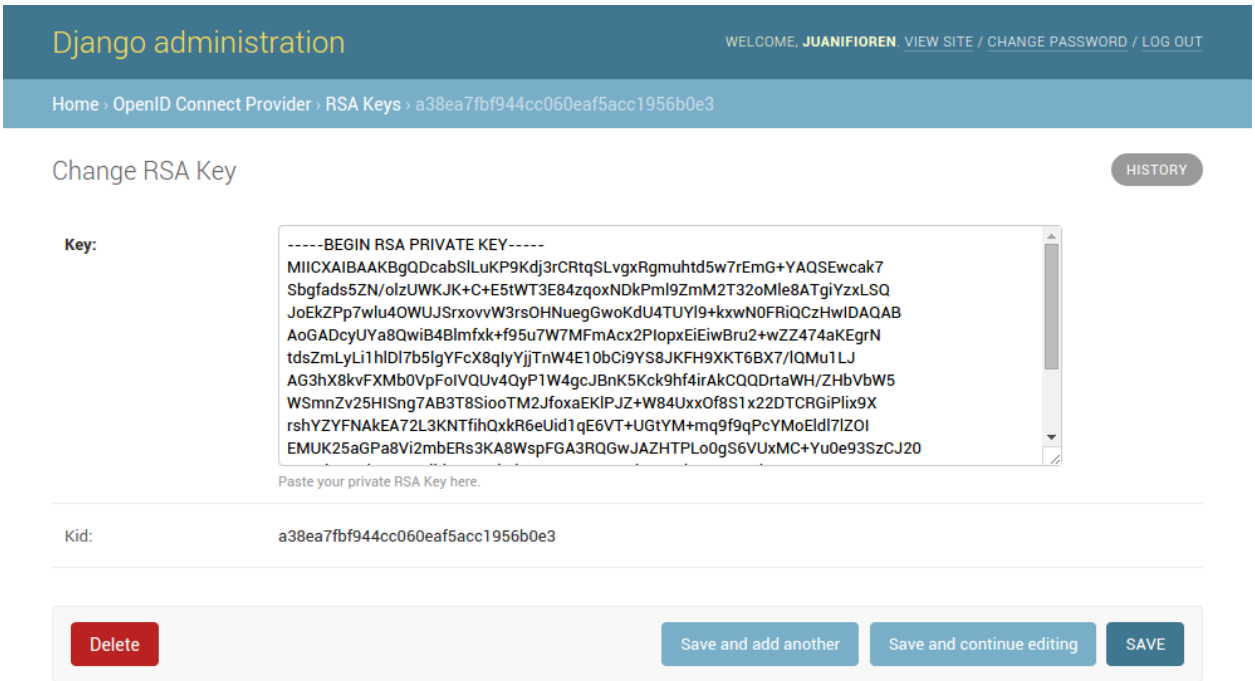
Read more about client creation in the OAuth2 spec

CHAPTER 3

Server Keys

Server RSA keys are used to sign/encrypt ID Tokens. These keys are stored in the `RSAKey` model. So the package will automatically generate public keys and expose them in the `jwks_uri` endpoint.

You can easily create them with the admin:



Or by using `python manage.py creatersakey` command.

Here is an example response from the `jwks_uri` endpoint:

```
GET /openid/jwks HTTP/1.1
Host: localhost:8000
```

(continues on next page)

(continued from previous page)


```
{
  "keys":[
    {
      "use":"sig",
      "e":"AQAB",
      "kty":"RSA",
      "alg":"RS256",
      "n":"3Gm0pS7ij_
↪SnY96wkbaki74MUYJrobXecO6xJhvmAEEhMHGpO0m4H2nbOWTf6Jc1FiiSvgvhObVk9xPOM6qMTQ5D5pfWZjNk99qDJXvAE4Im
↪",
      "kid":"a38ea7fbf944cc060eaf5acc1956b0e3"
    }
  ]
}
```

CHAPTER 4

Templates

Add your own templates files inside a folder named `templates/oidc_provider/`. You can copy the sample html files here and customize them with your own style.

authorize.html:

```
<h1>Request for Permission</h1>

<p>Client <strong>{{ client.name }}</strong> would like to access this information of␣
↪you ...</p>

<form method="post" action="{% url 'oidc_provider:authorize' %}">

    {% csrf_token %}

    {{ hidden_inputs }}

    <ul>
    {% for scope in scopes %}
        <li><strong>{{ scope.name }}</strong><br><i>{{ scope.description }}</i></li>
    {% endfor %}
    </ul>

    <input type="submit" value="Decline" />
    <input name="allow" type="submit" value="Authorize" />

</form>
```

error.html:

```
<h3>{{ error }}</h3>
<p>{{ description }}</p>
```

You can also customize paths to your custom templates by putting them in `OIDC_TEMPLATES` in the settings.

The following contexts will be passed to the `authorize` and `error` templates respectively:

```
# For authorize template
{
    'client': 'an instance of Client for the auth request',
    'hidden_inputs': 'a rendered html with all the hidden inputs needed for␣
↪AuthorizeEndpoint',
    'params': 'a dict containing the params in the auth request',
    'scopes': 'a list of scopes'
}

# For error template
{
    'error': 'string stating the error',
    'description': 'string stating description of the error'
}
```

CHAPTER 5

Scopes and Claims

This subset of OpenID Connect defines a set of standard Claims. They are returned in the UserInfo Response.

The package comes with a setting called `OIDC_USERINFO`, basically it refers to a function that will be called with `claims` (dict) and `user` (user instance). It returns the `claims` dict with all the claims populated.

List of all the `claims` keys grouped by scopes:

profile	email	phone	address
name	email	phone_number	formatted
given_name	email_verified	phone_number_verified	street_address
family_name			locality
middle_name			region
nickname			postal_code
preferred_username			country
profile			
picture			
website			
gender			
birthdate			
zoneinfo			
locale			
updated_at			

5.1 How to populate standard claims

Somewhere in your Django `settings.py`:

```
OIDC_USERINFO = 'myproject.oidc_provider_settings.userinfo'
```

Then inside your `oidc_provider_settings.py` file create the function for the `OIDC_USERINFO` setting:

```
def userinfo(claims, user):
    # Populate claims dict.
    claims['name'] = '{0} {1}'.format(user.first_name, user.last_name)
    claims['given_name'] = user.first_name
    claims['family_name'] = user.last_name
    claims['email'] = user.email
    claims['address']['street_address'] = '...'

    return claims
```

Now test an Authorization Request using these scopes `openid profile email` and see how user attributes are returned.

Note: Please **DO NOT** add extra keys or delete the existing ones in the `claims` dict. If you want to add extra claims to some scopes you can use the `OIDC_EXTRA_SCOPE_CLAIMS` setting.

5.2 How to add custom scopes and claims

The `OIDC_EXTRA_SCOPE_CLAIMS` setting is used to add extra scopes specific for your app. Is just a class that inherit from `oidc_provider.lib.claims.ScopeClaims`. You can create or modify scopes by adding this methods into it:

- `info_scopename` class property for setting the verbose name and description.
- `scope_scopename` method for returning some information related.

Let's say that you want add your custom `foo` scope for your OAuth2/OpenID provider. So when a client (RP) makes an Authorization Request containing `foo` in the list of scopes, it will be listed in the consent page (`templates/oidc_provider/authorize.html`) and then some specific claims like `bar` will be returned from the `/userinfo` response.

Somewhere in your Django `settings.py`:

```
OIDC_EXTRA_SCOPE_CLAIMS = 'yourproject.oidc_provider_settings.CustomScopeClaims'
```

Inside your oidc_provider_settings.py file add the following class:

```
from django.utils.translation import ugettext as _
from oidc_provider.lib.claims import ScopeClaims

class CustomScopeClaims(ScopeClaims):

    info_foo = (
        _(u'Foo'),
        _(u'Some description for the scope.'),
    )

    def scope_foo(self):
        # self.user - Django user instance.
        # self.userinfo - Dict returned by OIDC_USERINFO function.
        # self.scopes - List of scopes requested.
        # self.client - Client requesting this claims.
        dic = {
            'bar': 'Something dynamic here',
```

(continues on next page)

(continued from previous page)

```
        }

        return dic

    # If you want to change the description of the profile scope, you can redefine it.
    info_profile = (
        _(u'Profile'),
        _(u'Another description.'),
    )
```

Note: If a field is empty or `None` inside the dictionary you return on the `scope_scopename` method, it will be cleaned from the response.

CHAPTER 6

User Consent

The package store some information after the user grant access to some client. For example, you can use the `UserConsent` model to list applications that the user have authorized access. Like Google does here.

```
>>> from oidc_provider.models import UserConsent
>>> UserConsent.objects.filter(user__email='some@email.com')
[<UserConsent: Example Client - some@email.com>]
```

Note: the `UserConsent` model is not included in the admin.

6.1 Properties

- `user`: Django user object.
- `client`: Relying Party object.
- `expires_at`: Expiration date of the consent.
- `scope`: Scopes authorized.
- `date_given`: Date of the authorization.

CHAPTER 7

OAuth2 Server

Because OIDC is a layer on top of the OAuth 2.0 protocol, this package also gives you a simple but effective OAuth2 server that you can use not only for logging in your users on multiple platforms, but also to protect other resources you want to expose.

7.1 Protecting Views

Here we are going to protect a view with a scope called `read_books`:

```
from django.http import JsonResponse
from django.views.decorators.http import require_http_methods

from oidc_provider.lib.utils.oauth2 import protected_resource_view


@require_http_methods(['GET'])
@protected_resource_view(['read_books'])
def protected_api(request, *args, **kwargs):

    dic = {
        'protected': 'information',
    }

    return JsonResponse(dic, status=200)
```

7.2 Client Credentials Grant

The client can request an access token using only its client credentials (ID and SECRET) when the client is requesting access to the protected resources under its control, that have been previously arranged with the authorization server using the `client.scope` field.

Note: You can use Django admin to manually set the client scope or programmatically:

```
client.scope = ['read_books', 'add_books']
client.save()
```

This is how the request should look like:

```
POST /token HTTP/1.1
Host: localhost:8000
Authorization: Basic␣
↪eWZ3a3c0cWxtaHY0cToyVWE0QjVzRlhmZ3pNeXR5d1FqT01jNUsxYmpWeXhXeXRySVdsTmpQbld3\
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials
```

A successful access token response will like this:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
    "token_type"    : "Bearer",
    "access_token"  : "eyJhbGciOiJSUzI1NiIsImtpZCI6IjEifQ.eyJzY3AiOlsib3BlbmlkIiw...",
    "expires_in"    : 3600,
    "scope"         : "read_books add_books"
}
```

Token introspection can be used to validate access tokens requested with client credentials if the `OIDC_INTROSPECTION_VALIDATE_AUDIENCE_SCOPE` setting is `False`.

CHAPTER 8

Access Tokens

At the end of the login process, an access token is generated. This access token is the thing that is passed along with every API call to the openid connect server (e.g. userinfo endpoint) as proof that the call was made by a specific person from a specific app.

Access tokens generally have a lifetime of only a couple of hours. You can use `OIDC_TOKEN_EXPIRE` to set a custom expiration time that suits your needs.

8.1 Obtaining an Access Token

Go to the admin site and create a confidential client with `response_types = code` and `redirect_uri = http://example.org/`.

Open your browser and accept consent at:

```
http://localhost:8000/authorize?client_id=651462&redirect_uri=http://example.org/&
↪response_type=code&scope=openid email profile&state=123123
```

In the redirected URL you should have a `code` parameter included as query string:

```
http://example.org/?code=b9cedb346ee04f15ab1d3ac13da92002&state=123123
```

We use the `code` value to obtain `access_token` and `refresh_token`:

```
curl -X POST \
    -H "Cache-Control: no-cache" \
    -H "Content-Type: application/x-www-form-urlencoded" \
    "http://localhost:8000/token/" \
    -d "client_id=651462" \
    -d "client_secret=37b1c4ff826f8d78bd45e25bad75a2c0" \
    -d "code=b9cedb346ee04f15ab1d3ac13da92002" \
    -d "redirect_uri=http://example.org/" \
    -d "grant_type=authorization_code"
```

Example response:

```
{
    "access_token": "82b35f3d810f4cf49dd7a52d4b22a594",
    "token_type": "bearer",
    "expires_in": 3600,
    "refresh_token": "0bac2d80d75d46658b0b31d3778039bb",
    "id_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6..."
}
```

Then you can grab the access token and ask for user data by doing a GET request to the `/userinfo` endpoint:

```
curl -X GET \
    -H "Cache-Control: no-cache" \
    "http://localhost:8000/userinfo/?access_token=82b35f3d810f4cf49dd7a52d4b22a594"
```

8.2 Expiration and Refresh of Access Tokens

If you receive a `401 Unauthorized` status when using the access token, this probably means that your access token has expired.

The RP application can request a new access token by using the refresh token. Send a POST request to the `/token` endpoint with the following request parameters:

```
curl -X POST \
    -H "Cache-Control: no-cache" \
    -H "Content-Type: application/x-www-form-urlencoded" \
    "http://localhost:8000/token/" \
    -d "client_id=651462" \
    -d "client_secret=37b1c4ff826f8d78bd45e25bad75a2c0" \
    -d "grant_type=refresh_token" \
    -d "refresh_token=0bac2d80d75d46658b0b31d3778039bb"
```

CHAPTER 9

Session Management

The OpenID Connect Session Management 1.0 specification complements the core specification by defining how to monitor the End-User's login status at the OpenID Provider on an ongoing basis so that the Relying Party can log out an End-User who has logged out of the OpenID Provider.

9.1 Setup

Somewhere in your Django `settings.py`:

```
MIDDLEWARE_CLASSES = [
    ...
    'oidc_provider.middleware.SessionManagementMiddleware',
]

OIDC_SESSION_MANAGEMENT_ENABLE = True
```

If you're in a multi-server setup, you might also want to add `OIDC_UNAUTHENTICATED_SESSION_MANAGEMENT_KEY` to your settings and set it to some random but fixed string. While authenticated clients have a session that can be used to calculate the browser state, there is no such thing for unauthenticated clients. Hence this value. By default a value is generated randomly on startup, so this will be different on each server. To get a consistent value across all servers you should set this yourself.

9.2 Example RP iframe

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="ISO-8859-1">
    <title>RP Iframe</title>
</head>
```

(continues on next page)

(continued from previous page)

```
<body onload="javascript:startChecking()">
    <iframe id="op-iframe" src="http://localhost:8000/check-session-iframe/"␣
↪frameborder="0" width="0" height="0"></iframe>
</body>
<script>
    var targetOP = "http://localhost:8000";

    window.addEventListener("message", receiveMessage, false);

    function startChecking() {
              checkStatus();
              setInterval('checkStatus()', 1000*60); // every 60 seconds
    }

    function checkStatus() {
              var clientId = '';
              var sessionState = '';
              var data = clientId + ' ' + sessionState;
              document.getElementById('op-iframe').contentWindow.postMessage(data,␣
↪targetOP);
    }

    function receiveMessage(event) {
        if (event.origin !== targetOP) {
            // Origin did not come from the OP.
            return;
        }
        if (event.data === 'unchanged') {
            // User is still logged in to the OP.
        } else if (event.data === 'changed') {
            // Perform re-authentication with prompt=none to obtain the current␣
↪session state at the OP.
        } else {
            // Error.
            console.log('Something goes wrong!');
        }
    }
</script>
</html>
```

9.3 RP-Initiated Logout

An RP can notify the OP that the End-User has logged out of the site, and might want to log out of the OP as well. In this case, the RP, after having logged the End-User out of the RP, redirects the End-User's User Agent to the OP's logout endpoint URL.

This URL is normally obtained via the `end_session_endpoint` element of the OP's Discovery response.

Parameters that are passed as query parameters in the logout request:

- **`id_token_hint`** Previously issued ID Token passed to the logout endpoint as a hint about the End-User's current authenticated session with the Client.
- **`post_logout_redirect_uri`** URL to which the RP is requesting that the End-User's User Agent be redirected after a logout has been performed.

- **state** OPTIONAL. Opaque value used by the RP to maintain state between the logout request and the callback to the endpoint specified by the `post_logout_redirect_uri` query parameter.

Example redirect:

```
http://localhost:8000/end-session/?id_token_hint=eyJhbGciOiJSUzI1NiIsImtpZCI6ImQwM...&
↪post_logout_redirect_uri=http://rp.example.com/logged-out/&state=c91c03ea6c46a86
```

CHAPTER 10

Token Introspection

The OAuth 2.0 Authorization Framework extends its scope with many other speficications. One of these is the OAuth 2.0 Token Introspection (RFC 7662) which defines a protocol that allows authorized protected resources to query the authorization server to determine the set of metadata for a given token that was presented to them by an OAuth 2.0 client.

10.1 Client Setup

In order to enable this feature, some configurations must be performed in the `Client`.

- The scope key:`token_introspection` must be added to the client's scope.

If `OIDC_INTROSPECTION_VALIDATE_AUDIENCE_SCOPE` is set to `True` then:

- The `client_id` must be added to the client's scope.

10.2 Introspection Endpoint

The introspection endpoint `(/introspect)` is an OAuth 2.0 endpoint that takes a parameter representing an OAuth 2.0 token and returns a JSON document representing the meta information surrounding the token.

The introspection endpoint its called using an HTTP POST request with parameters sent as *"application/x-www-form-urlencoded"* and **Basic authentication** (`base64(client_id:client_secret)`).

Parameters:

- **`token`** REQUIRED. The string value of an `access_token` previously issued.

Example request:

```
curl -X POST \
http://localhost:8000/introspect \
-H 'Authorization: Basic NDgwNTQ2OmIxOGIyODVmY2E5N2Fm' \
```

(continues on next page)

(continued from previous page)

```
-H 'Content-Type: application/x-www-form-urlencoded' \
-d token=6dd4b859706944848183d26f2fcb99c6
```

Example Response:

```
{
    "aud": "480546",
    "sub": "1",
    "exp": 1538971676,
    "iat": 1538971076,
    "iss": "http://localhost:8000",
    "active": true,
    "client_id": "480546"
}
```

10.3 Introspection Endpoint Errors

In case of error, the Introspection Endpoint will return a JSON document with the key `active: false`

Example Error Response:

```
{
    "active": "false"
}
```

CHAPTER 11

Settings

Customize django-oidc-provider so that it fits your project's needs.

11.1 OIDC_LOGIN_URL

OPTIONAL. `str`. Used to log the user in. By default Django's `LOGIN_URL` will be used. Read more in the Django docs

`str`. Default is `/accounts/login/` (Django's `LOGIN_URL`).

11.2 SITE_URL

OPTIONAL. `str`. The OP server url.

If not specified, it will be automatically generated using `request.scheme` and `request.get_host()`.

For example `http://localhost:8000`.

11.3 OIDC_AFTER_USERLOGIN_HOOK

OPTIONAL. `str`. A string with the location of your function. Provide a way to plug into the process after the user has logged in, typically to perform some business logic.

Default is:

```
def default_hook_func(request, user, client):
    return None
```

Return `None` if you want to continue with the flow.

The typical situation will be checking some state of the user or maybe redirect him somewhere. With `request` you have access to all OIDC parameters. Remember that if you redirect the user to another place then you need to take him back to the authorize endpoint (use `request.get_full_path()` as the value for a "next" parameter).

11.4 OIDC_AFTER_END_SESSION_HOOK

OPTIONAL. `str`. A string with the location of your function. Provide a way to plug into the log out process just before calling Django's log out function, typically to perform some business logic.

Default is:

```
def default_after_end_session_hook(request, id_token=None, post_logout_redirect_
↪uri=None, state=None, client=None, next_page=None):
    return None
```

Return `None` if you want to continue with the flow.

11.5 OIDC_CODE_EXPIRE

OPTIONAL. `int`. Code object expiration after been delivered.

Expressed in seconds. Default is `60*10`.

11.6 OIDC_EXTRA_SCOPE_CLAIMS

OPTIONAL. `str`. A string with the location of your class. Default is `oidc_provider.lib.claims.ScopeClaims`.

Used to add extra scopes specific for your app. OpenID Connect RP's will use scope values to specify what access privileges are being requested for Access Tokens.

Read more about how to implement it in *Scopes and Claims* section.

11.7 OIDC_IDTOKEN_INCLUDE_CLAIMS

OPTIONAL. `bool`. If enabled, id_token will include standard claims of the user (email, first name, etc.).

Default is `False`.

11.8 OIDC_IDTOKEN_EXPIRE

OPTIONAL. `int`. ID Token expiration after been delivered.

Expressed in seconds. Default is `60*10`.

11.9 OIDC_IDTOKEN_PROCESSING_HOOK

OPTIONAL. `str` or `(list, tuple)`.

A string with the location of your function hook or `list` or `tuple` with hook functions. Here you can add extra dictionary values specific for your app into id_token.

The `list` or `tuple` is useful when you want to set multiple hooks, i.e. one for permissions and second for some special field.

The hook function receives following arguments:

- `id_token`: the ID token dictionary which contains at least the basic claims (`iss`, `sub`, `aud`, `exp`, `iat`, `auth_time`), but may also contain other claims. If several processing hooks are configured, then the claims of the previous hook are also present in the passed dictionary.
- `user`: User object of the authenticating user,
- `token`: the Token object created for the authentication request, and
- `request`: Django request object of the authentication request.

The hook function should return the modified ID token as dictionary.

Note: It is a good idea to add `**kwargs` to the hook function argument list so that the hook function will work even if new arguments are added to the hook function call signature.

Default is:

```
def default_idtoken_processing_hook(id_token, user, token, request, **kwargs):

    return id_token
```

11.10 OIDC_IDTOKEN_SUB_GENERATOR

OPTIONAL. `str`. A string with the location of your function. `sub` is a locally unique and never reassigned identifier within the Issuer for the End-User, which is intended to be consumed by the Client.

The function receives a `user` object and returns a unique `string` for the given user.

Default is:

```
def default_sub_generator(user):

    return str(user.id)
```

11.11 OIDC_INTROSPECTION_PROCESSING_HOOK

OPTIONAL. `str` or `(list, tuple)`.

A string with the location of your function hook or `list` or `tuple` with hook functions. Here you can add extra dictionary values specific to your valid response value for token introspection.

The function receives an `introspection_response` dictionary, a `client` instance and an `id_token` dictionary.

If the token is generated by client_credentials grant then `id_token` is `None`.

Default is:

```
def default_introspection_processing_hook(introspection_response, client, id_token):

    return introspection_response
```

11.12 OIDC_INTROSPECTION_VALIDATE_AUDIENCE_SCOPE

OPTIONAL `bool`

A flag which toggles whether the audience is matched against the client resource scope when calling the introspection endpoint.

Must be `False` to support introspecting client_crendentials tokens.

Default is `True`.

11.13 OIDC_SESSION_MANAGEMENT_ENABLE

OPTIONAL. `bool`. Enables OpenID Connect Session Management 1.0 in your provider. See the *Session Management* section.

Default is `False`.

11.14 OIDC_UNAUTHENTICATED_SESSION_MANAGEMENT_KEY

OPTIONAL. Supply a fixed string to use as browser-state key for unauthenticated clients. See the *Session Management* section.

Default is a string generated at startup.

11.15 OIDC_SKIP_CONSENT_EXPIRE

OPTIONAL. `int`. How soon User Consent expires after being granted.

Expressed in days. Default is `30*3`.

11.16 OIDC_TOKEN_EXPIRE

OPTIONAL. `int`. Token object (access token) expiration after being created.

Expressed in seconds. Default is `60*60`.

11.17 OIDC_USERINFO

OPTIONAL. `str`. A string with the location of your function. See the *Scopes and Claims* section.

The function receives a `claims` dictionary with all the standard claims and `user` instance. Must returns the `claims` dict again.

Example usage:

```
def userinfo(claims, user):

    claims['name'] = '{0} {1}'.format(user.first_name, user.last_name)
    claims['given_name'] = user.first_name
    claims['family_name'] = user.last_name
    claims['email'] = user.email
    claims['address']['street_address'] = '...'

    return claims
```

Note: Please **DO NOT** add extra keys or delete the existing ones in the `claims` dict. If you want to add extra claims to some scopes you can use the `OIDC_EXTRA_SCOPE_CLAIMS` setting.

11.18 OIDC_GRANT_TYPE_PASSWORD_ENABLE

OPTIONAL. A boolean whether to allow the Resource Owner Password Credentials Grant. https://tools.ietf.org/html/rfc6749#section-4.3

Important: From the specification: "Since this access token request utilizes the resource owner's password, the authorization server **MUST** protect the endpoint against brute force attacks (e.g., using rate-limitation or generating alerts)."

There are many ways to implement brute force attack prevention. We cannot decide what works best for you, so you will have to implement a solution for this that suits your needs.

11.19 OIDC_TEMPLATES

OPTIONAL. A dictionary pointing to templates for authorize and error pages. Default is:

```
{
    'authorize': 'oidc_provider/authorize.html',
    'error': 'oidc_provider/error.html'
}
```

See the *Templates* section.

The templates that are not specified here will use the default ones.

CHAPTER 12

Signals

Use signals in your application to get notified when some actions occur.

For example:

```
from django.dispatch import receiver

from oidc_provider.signals import user_decline_consent


@receiver(user_decline_consent)
def my_callback(sender, **kwargs):
    print(kwargs)
    print('Ups! Some user has declined the consent.')
```

12.1 user_accept_consent

Sent when a user accept the authorization page for some client.

12.2 user_decline_consent

Sent when a user decline the authorization page for some client.

CHAPTER 13

Examples

13.1 Pure JS client using Implicit Flow

Testing OpenID Connect flow can be as simple as putting one file with a few functions on the client and calling the provider. Let me show.

01. Setup the provider

You can use the example project code to run your OIDC Provider at `localhost:8000`.

Go to the admin site and create a public client with a response_type `id_token token` and a redirect_uri `http://localhost:3000`.

Note: Remember to create at least one **RSA Key** for the server with `python manage.py creatersakey`

02. Create the client

As relying party we are going to use a JS library created by Nat Sakimura. Here is the article.

index.html:

```
<!DOCTYPE html>
<html>
<head>

    <title>OIDC RP</title>

</head>
<body>

    <center>
        <h1>OpenID Connect RP Example</h1>
        <button id="login-button">Login</button>
    </center>
```

(continues on next page)

(continued from previous page)

```
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.2.2/jquery.min.js"></
↪script>
    <script src="https://www.sakimura.org/test/openidconnect.js"></script>

    <script type="text/javascript">
    $(function() {
        var clientInfo = {
            client_id : '',
            redirect_uri : 'http://localhost:3000'
        };

        OIDC.setClientInfo(clientInfo);

        var providerInfo = OIDC.discover('http://localhost:8000');

        OIDC.setProviderInfo(providerInfo);
        OIDC.storeInfo(providerInfo, clientInfo);

        // Restore configuration information.
        OIDC.restoreInfo();

        // Get Access Token
        var token = OIDC.getAccessToken();

        // Make userinfo request using access_token.
        if (token !== null) {
            $.get('http://localhost:8000/userinfo/?access_token='+token, function(␣
↪data ) {
                alert('USERINFO: '+ JSON.stringify(data));
            });
        }

        // Make an authorization request if the user click the login button.
        $('#login-button').click(function (event) {
            OIDC.login({
                scope : 'openid profile email',
                response_type : 'id_token token'
            });
        });
    });
    </script>

</body>
</html>
```

Note: Remember that you must set your client_id (line 21).

03. Make an authorization request

By clicking the login button an authorization request has been made to the provider. After you accept it, the provider will redirect back to your previously registered `redirect_uri` with all the tokens requested.

04. Requesting user information

Now having the access_token in your hands you can request the user information by making a request to the /

`userinfo` endpoint of the provider.

In this example we display information in the alert box.

CHAPTER 14

Contribute

We love contributions, so please feel free to fix bugs, improve things, provide documentation. These are the steps:

- Create an issue and explain your feature/bugfix.
- Wait collaborators comments.
- Fork the project and create new branch from *develop*.
- Make your feature addition or bug fix.
- Add tests and documentation if needed.
- Create pull request for the issue to the *develop* branch.
- Wait collaborators reviews.

14.1 Running Tests

Use tox for running tests in each of the environments, also to run coverage and flake8 among:

```
# Run all tests.
$ tox

# Run with Python 3.5 and Django 2.0.
$ tox -e py35-django20

# Run single test file on specific environment.
$ tox -e py35-django20 tests/cases/test_authorize_endpoint.py
```

We also use travis to automatically test every commit to the project.

14.2 Improve Documentation

We use Sphinx for generate this documentation. I you want to add or modify something just:

- Install Sphinx (`pip install sphinx`) and the auto-build tool (`pip install sphinx-autobuild`).
- Move inside the docs folder. `cd docs/`
- Generate and watch docs by running `sphinx-autobuild . _build/`.
- Open `http://127.0.0.1:8000` in a browser.

CHAPTER 15

Changelog

All notable changes to this project will be documented in this file.

15.1 Unreleased

15.2 0.7.0

2018-10-17

- Added: support multiple response types per client.
- Added: make version available in code.
- Added: token introspection docs.
- Changed: drop support for Django versions lower than 1.11.
- Changed: create RSA key command. Increment key size to 2048.
- Fixed: OIDC_IDTOKEN_INCLUDE_CLAIMS used with custom claims setting.
- Fixed: bug in prompt parameter (with space-separated values).

15.3 0.6.2

2018-08-03

- Added: support introspection on client credentials tokens.
- Changed: accept lowercase “bearer” in Authorization header.
- Fixed: ScopeClaims class.
- Fixed: code is not zip safe.

15.4 0.6.1

2018-07-10

- Added: token instrospection endpoint support (RFC7662).
- Added: request in password grant authenticate call.
- Changed: dropping support for Django versions before 1.8.
- Changed: pass token and request to OIDC_IDTOKEN_PROCESSING_HOOK.
- Fixed: CORS OPTIONS request blocked on userinfo request.
- Fixed: settings to support falsy valued overrides.
- Fixed: token introspection "aud" and "client_id" response.
- Fixed: Token Model str() crashes when using client credentials grant.

15.5 0.6.0

2018-04-13

- Added: OAuth2 grant_type client_credentials support.
- Added: pep8 compliance and checker.
- Added: Setting OIDC_IDTOKEN_INCLUDE_CLAIMS supporting claims inside id_token.
- Changed: Test suit now uses pytest.
- Fixed: Infinite callback loop in the check-session iframe.

15.6 0.5.3

2018-03-09

- Fixed: Update project to support Django 2.0

15.7 0.5.2

2017-08-22

- Fixed: infinite login loop if "prompt=login" (#198)
- Fixed: Django 2.0 deprecation warnings (#185)

15.8 0.5.1

2017-07-11

- Changed: Documentation template changed to Read The Docs.
- Fixed: install_requires has not longer pinned versions.

- Fixed: Removed infinity loop during authorization stage when prompt=login has been send.
- Fixed: Changed prompt handling as set of options instead of regular string.
- Fixed: Redirect URI must match exactly with given in query parameter.
- Fixed: Stored user consent are useful for public clients too.
- Fixed: documentation for custom scopes handling.
- Fixed: Scopes during refresh and code exchange are being taken from authorization request and not from query parameters.

15.9 0.5.0

2017-05-18

- Added: signals when user accept/decline the authorization page.
- Added: OIDC_AFTER_END_SESSION_HOOK setting for additional business logic.
- Added: feature granttype password.
- Added: require_consent and reuse_consent are added to Client model.
- Changed: OIDC_SKIP_CONSENT_ALWAYS and OIDC_SKIP_CONSENT_ENABLE are removed from settings.
- Fixed: timestamps with unixtime (instead of django timezone).
- Fixed: field refresh_token cannot be primary key if null.
- Fixed: create_uri_exceptions are now being logged at Exception level not DEBUG.

15.10 0.4.4

2016-11-29

- Fixed: Bug in Session Management middleware when using Python 3.
- Fixed: Translations handling.

15.11 0.4.3

2016-11-02

- Added: Session Management 1.0 support.
- Added: post_logout_redirect_uris into admin.
- Changed: Package url names.
- Changed: Rename /logout/ url to /end-session/.
- Fixed: bug when trying authorize with response_type id_token without openid scope.

15.12 0.4.2

2016-10-13

- Added: support for client redirect URIs with query strings.
- Fixed: bug when generating secret_key value using admin.
- Changed: client is available to OIDC_EXTRA_SCOPE_CLAIMS implementations via self.client.
- Changed: the constructor signature for ScopeClaims has changed, it now is called with the Token as its single argument.

15.13 0.4.1

2016-10-03

- Changed: update pyjwkest to version 1.3.0.
- Changed: use Cryptodome instead of Crypto lib.

15.14 0.4.0

2016-09-12

- Added: support for Hybrid Flow.
- Added: new attributes for Clients: Website url, logo, contact email, terms url.
- Added: polish translations.
- Added: examples section in documentation.
- Fixed: CORS in discovery and userinfo endpoint.
- Fixed: client type public bug when created using the admin.
- Fixed: missing OIDC_TOKEN_EXPIRE setting on implicit flow.

15.15 0.3.7

2016-08-31

- Added: support for Django 1.10.
- Added: initial translation files (ES, FR).
- Added: support for at_hash parameter.
- Fixed: empty address dict in userinfo response.

15.16 0.3.6

2016-07-07

- Changed: OIDC_USERINFO setting.

15.17 0.3.5

2016-06-21

- Added: field date_given in UserConsent model.
- Added: verbose names to all model fields.
- Added: customize scopes names and descriptions on authorize template.
- Changed: OIDC_EXTRA_SCOPE_CLAIMS setting.

15.18 0.3.4

2016-06-10

- Changed: Make SITE_URL setting optional.
- Fixed: Missing migration.

15.19 0.3.3

2016-05-03

- Fixed: Important bug with PKCE and form submit in Auth Request.

15.20 0.3.2

2016-04-26

- Added: choose type of client on creation.
- Added: implement Proof Key for Code Exchange by OAuth Public Clients.
- Added: support for prompt parameter.
- Added: support for different client JWT tokens algorithm.
- Fixed: not auto-approve requests for non-confidential clients (publics).

15.21 0.3.1

2016-03-09

- Fixed: response_type was not being validated (OpenID request).

15.22 0.3.0

2016-02-23

- Added: support OAuth2 requests.

- Added: decorator for protecting views with OAuth2.
- Added: setting OIDC_IDTOKEN_PROCESSING_HOOK.

15.23 0.2.5

2016-02-03

- Added: Setting OIDC_SKIP_CONSENT_ALWAYS.
- Changed: Removing OIDC_RSA_KEY_FOLDER setting. Moving RSA Keys to the database.
- Changed: Update pyjwkest to version 1.1.0.
- Fixed: Nonce parameter missing on the decide form.
- Fixed: Set Allow-Origin header to jwks endpoint.

15.24 0.2.4

2016-01-20

- Added: Auto-generation of client ID and SECRET using the admin.
- Added: Validate nonce parameter when using Implicit Flow.
- Fixed: generating RSA key by ignoring value of OIDC_RSA_KEY_FOLDER.
- Fixed: make OIDC_AFTER_USERLOGIN_HOOK and OIDC_IDTOKEN_SUB_GENERATOR to be lazy imported by the location of the function.
- Fixed: problem with a function that generate urls for the /.well-known/openid-configuration/ endpoint.

15.25 0.2.3

2016-01-06

- Added: Make user and client unique on UserConsent model.
- Added: Support for URL's without end slash.
- Changed: Upgrade pyjwkest to version 1.0.8.
- Fixed: String format error in models.
- Fixed: Redirect to non http urls fail (for Mobile Apps).

15.26 0.2.1

2015-10-21

- Added: refresh token flow.
- Changed: upgrade pyjwkest to version >= 1.0.6.
- Fixed: Unicode error in Client model.

- Fixed: Bug in creatersakey command (when using Python 3).
- Fixed: Bug when updating pyjwkest version.

15.27 0.2.0

2015-09-25

- Changed: UserInfo model was removed. Now you can add your own model using OIDC_USERINFO setting.
- Fixed: ID token does NOT contain kid.

15.28 0.1.2

2015-08-04

- Added: add token_endpoint_auth_methods_supported to discovery.
- Fixed: missing commands folder in setup file.

15.29 0.1.1

2015-07-31

- Added: sending access_token as query string parameter in UserInfo Endpoint.
- Added: support HTTP Basic client authentication.
- Changed: use models setting instead of User.
- Fixed: in python 2: “aud” and “nonce” parameters didn’t appear in id_token.

15.30 0.1.0

2015-07-17

- Added: now id tokens are signed/encrypted with RS256.
- Added: command for easily generate random RSA key.
- Added: jwks uri to discovery endpoint.
- Added: id_token_signing_alg_values_supported to discovery endpoint.
- Fixed: nonce support for both Code and Implicit flow.

15.31 0.0.7

2015-07-06

- Added: support for Python 3.

- Added: way of remember user consent and skipt it (OIDC_SKIP_CONSENT_ENABLE).
- Added: setting OIDC_SKIP_CONSENT_EXPIRE.
- Changed: now OIDC_EXTRA_SCOPE_CLAIMS must be a string, to be lazy imported.

15.32 0.0.6

2015-06-16

- Added: better naming for models in the admin.
- Changed: now tests run without the need of a project configured.
- Fixed: error when returning address_formatted claim.

15.33 0.0.5

2015-05-09

- Added: support for Django 1.8.
- Fixed: validation of scope in UserInfo endpoint.

15.34 0.0.4

2015-04-22

- Added: initial migrations.
- Fixed: important bug with id_token when using implicit flow.
- Fixed: validate Code expiration in Auth Code Flow.
- Fixed: validate Access Token expiration in UserInfo endpoint.

15.35 0.0.3

2015-04-15

- Added: normalize gender field in UserInfo.
- Changed: make address_formatted a property inside UserInfo.
- Fixed: important bug in claims response.

15.36 0.0.2

2015-03-26

- Added: setting OIDC_AFTER_USERLOGIN_HOOK.
- Fixed: tests failing because an incorrect tag in one template.

15.37 0.0.1

2015-03-13

- Added: provider Configuration Information endpoint.
- Added: setting OIDC_IDTOKEN_SUB_GENERATOR.
- Changed: now use setup in OIDC_EXTRA_SCOPE_CLAIMS setting.

15.38 0.0.0

2015-02-26

CHAPTER 16

Indices and tables

- genindex
- modindex
- search